Instituto Superior Técnico

Programação de Sistemas

Relatório do Projeto

Grupo 43:
Inês Ferreira (90395)
Eduardo Cunha (90060)

2020/2021 - 2º Semestre
6.06.2021

Introdução

O sistema a ser implementado consiste em três subsistemas que comunicam entre si - aplicações, servidores locais, e um servidor autenticador. As aplicações estão ligadas a grupo num servidor local, dento do qual podem requisitar vários pedidos (através do teclado) relativos aos pares chave-valor. No servidor local é possível criar e eliminar grupos e mostrar informação sobre os mesmos. O conjunto de todos os grupos de todos os servidores locais está guardado no servidor autenticador, juntamente com os segredos de cada grupo, que permitem as aplicações ligarem-se.

Ao longo deste relatório iremos explicar em detalhe como é que as várias operações foram implementadas, nomeadamente em termos de comunicação entre os vários subsistemas.

Também mostraremos as várias estruturas de dados utilizadas, quer para guardar os pares chave-valor, quer para guardar os pares grupo-segredo.

Arquitetura

O esquema abaixo representa uma imagem geral do nosso sistema.

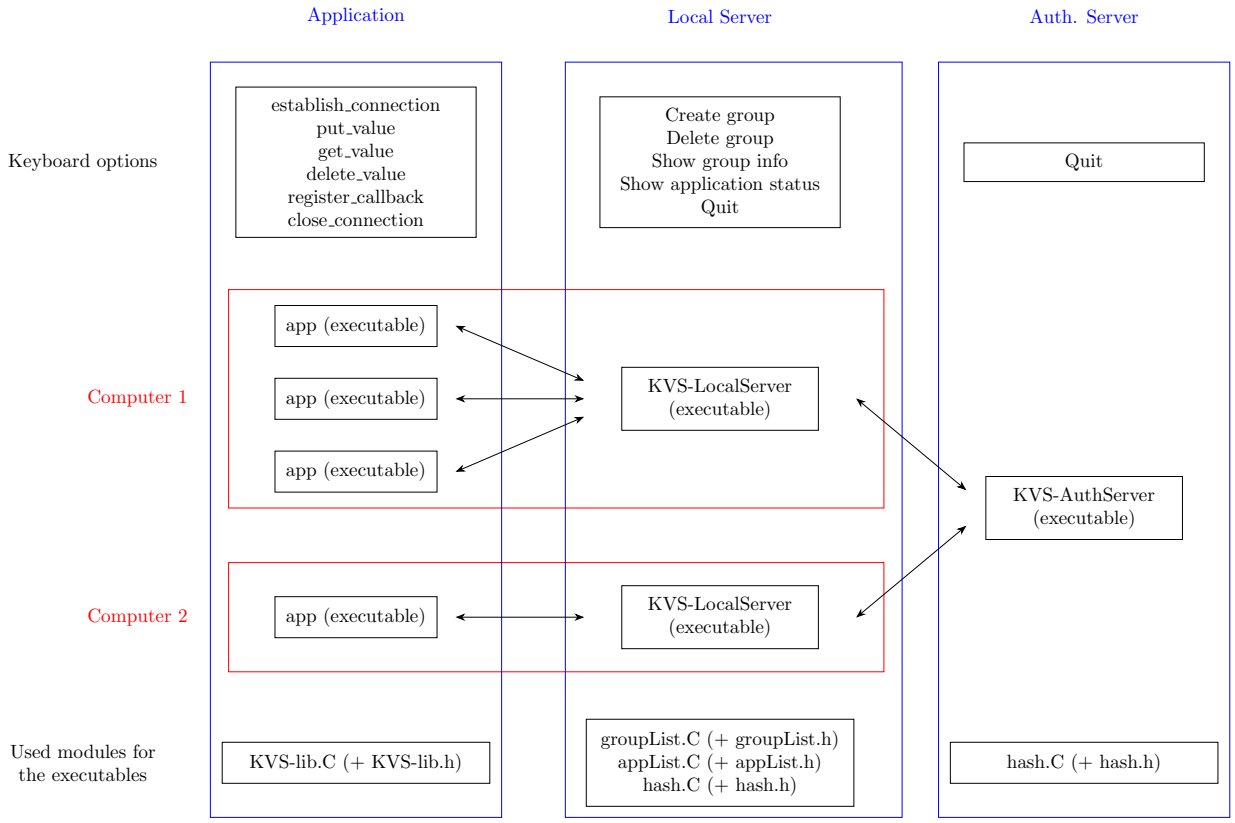

Figura 1: Arquitetura geral do sistema e dos subsistemas.

Cada retângulo a azul representa um dos três subsistemas que interagem entre si. No topo de cada subsistema temos as opções que o utilizador pode colocar na linha de comandos de cada um dos executáveis. Note-se que para o servidor local e para o servidor autenticador se adicionou a opção "Quit"que se assemelha à opção "close_connection"da aplicação.

Na zona central é descrito como os vários executáveis interagem entre si.

Por fim, na zona de baixo, temos os vários módulos / ficheiros auxiliares que são utilizados para criar cada um dos executáveis. Estes serão explicados em detalhe na secção seguinte.

Estrutura do código

Módulos

O código contém três ficheiros principais que produzem três executáveis, relativos ao servidor autenticador (*KVS-AuthServer*), ao servidor local (*KVS-LocalServer*) e à aplicação (*app*). Existem ficheiros *.c* com os mesmos nomes do que os executáveis que contêm as funções *main*. Existem outros ficheiros *.c* e *.h* (módulos) com declarações e definições de funções que são usadas pelos ficheiros já referidos. A aplicação utiliza as funções da biblioteca *KVS-lib* definidas nos ficheiros *.c* e *.h* com o mesmo nome. O servidor autenticador utiliza os ficheiros *hash.c* e *hash.h* e, por fim, o servidor local utiliza, para além dos ficheiros também utilizados pelo servidor autenticador, os ficheiros *groupList.c*, *groupList.h*, *appList.c* e *appList.h*. Para poder explicar melhor o conteúdo de cada um dos módulos vamos começar por explicar as estruturas de dados que foram utilizadas em cada um dos servidores e na aplicação.

Estruturas de dados

Aplicação

A aplicação apenas tem uma lista ligada em que cada nó (uma estrutura chamada *cb_info*) contém um *array* de *char* (chave), um ponteiro para uma função (função de *callback* correspondente à chave) e um ponteiro para o próximo nó da lista. A função obrigatoriamente recebe um ponteiro para *char* e retorna *void*. Decidimos guardar também a função de *callback* na lista para que, se no futuro criarmos um *main* diferente (*app.c*) com várias funções de *callback*, estas possam ser associadas a diferentes chaves. A esta lista de pares chave-função vamos chamar *WatchList*.

Servidor Local

No servidor local, toda a informação relevante está guardada numa lista ligada de estruturas *Group*. Cada *Group* tem um nome (*array* de *char*), um ponteiro para uma *HashTable* (onde vão ser guardados os pares chave-valor), uma lista ligada de estruturas *App*, e um ponteiro para o próximo nó. Manter a lista de aplicações em cada grupo permite, entre outras funcionalidades, obrigar as aplicações que ainda estão conectadas a desconectar-se quando um grupo é eliminado e saber que aplicações notificar quando uma chave de um grupo é alterada.

A estrutura *App* contém:

- PID - É o identificador da aplicação porque é único.
- *File Descriptor*: para saber para onde enviar a informação e para poder ser fechado quando se sai do servidor.
- *File Descriptor Para O Callback*: um segundo *File Descriptor* para comunicações relativas às funções de *callback* e para o caso especial em que o servidor é fechado sem que todas as aplicações se tenham desconectado.
- *bool Flag* (*isClosed*): um valor que permite saber se a aplicação ainda está conectada ou não. Útil para saber como calcular o tempo de ligação, para enviar (ou não) mensagens (a propósito das chaves que foram mudadas e que têm uma função de *callback* associada) e para saber se é preciso fechar os *File Descriptors* aquando da eliminação de grupos ou do término do servidor.
- Estrutura de tempo *start* para imprimir o instante do início da conexão da aplicação.
- Estrutura de tempo *stop* para imprimir o instante de fecho da conexão da aplicação.
- *double delta_t*: intervalo de tempo de conexão da aplicação.
- Uma lista ligada de estruturas *wlist* (*WatchList*): contém as chaves de uma dada aplicação que têm uma função de *callback* associada. A lista é necessária para saber se é preciso notificar a aplicação para esta chamar a função de *callback*, aquando da mudança de chave.
- Ponteiro para o próximo nó.

A estrutura *wlist* apenas contém um *array* de *char* (chave) e um ponteiro para o próximo nó.

Por fim, o servidor autenticador tem um ponteiro para uma HashTable onde são guardados os pares grupo-segredo e tem uma lista ligada de estruturas *LocalSvrData*. Estas apenas têm um inteiro que corresponde ao *File Descriptor* de cada servidor que está conectado. Assim, é possível fechar os *File Descriptors* correspondentes às ligações com os servidores locais caso o servidor autenticador feche antes de todas as ligações estarem terminadas.

Agora que temos todas as estruturas de dados descritas, é fácil fazer a correspondência com os ficheiros. Todas as funções correspondentes a criar, inserir dados, eliminar dados e libertar memória de tabelas estão presentes nos ficheiros *hash.c* e *hash.h*, juntamente com mais funções auxiliares.

O mesmo acontece para a lista ligada de aplicações que vai ser utilizada nos grupos e os ficheiros *applist.c* e *appList.h*.

Para além das funções relativas à lista ligada dos grupos, nos ficheiros *groupList.c* e *groupList.h* também se encontram algumas funções correspondentes a criar e eliminar grupos no servidor autenticador que são utilizadas pelo servidor local.

Por fim, é importante referir que todas as *strings* utilizadas no programa, quer sejam provenientes do teclado, ou guardadas em memória, têm um tamanho máximo de 100. Isto permite definir os *array* de *char* com o tamanho fixo escolhido, tornando a alocação de memória mais simples.

HashTable

A HashTable vai ser utilizada pelo servidor local, para guardar os pares chave-valor, e pelo servidor autenticador, para guardar os pares grupo-segredo. Note-se que em ambos os casos os dois atributos são *strings* que podem ser guardadas como *array* de *char*. Assim, podemos utilizar a mesma implementação da tabela para os dois servidores. Definimos também um tamanho máximo de 50000 entradas embora, como se vai ver, de certa forma o tamanho da tabela é ilimitado.

A *HashFunction* que foi utilizada é a soma de cada um dos caracteres da chave / grupo convertidos para inteiros, escalada de forma a dar um número máximo igual à capacidade da tabela.

A estrutura *HashTable* em si contém um *array* de ponteiros para outra estrutura (*items*), um *array* de ponteiros para listas ligadas, o tamanho máximo e o número de elementos. A estrutura *items* contém dois *array* de *char* (par chave-valor). Com esta estrutura é possível ter uma tabela que, para cada índice (valor dado pela *HashFunction*), tem uma entrada e uma lista ligada. A lista ligada serve para, no caso de duas chaves terem o mesmo output da *HashFunction*, uma delas ser guardada na tabela em si e a outra ser guardada na lista ligada. Assim, observa-se que, se muitas chaves tiverem o mesmo índice, existe sempre forma de as guardar, tornando o tamanho da tabela "ilimitado".

Abaixo encontra-se um esquema da tabela retirado de `https://en.wikipedia.org/wiki/Hash_table` que representa como os valores são guardados.

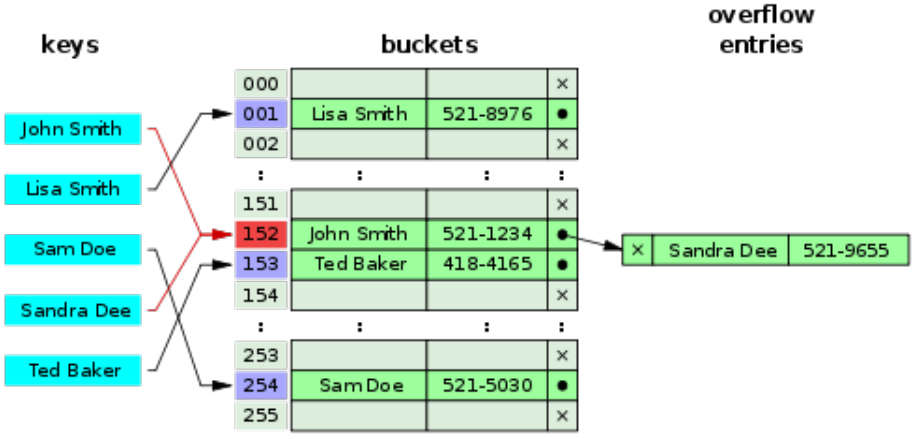

Figura 2: Esquema da *HashTable*. Os *overflow entries* são os valores que são guardados na lista ligada correspondente a um determinado índice e *buckets* corresponde à própria tabela em si,

Organização das *Threads*

De seguida iremos explicar, para cada um dos ficheiros com *main functions*, as diferentes *threads* que existem, como são criadas e destruídas, e a sua funcionalidade.

app.c

A aplicação tem 3 *threads* no total. Um esquema de como elas são criadas e as suas funções está presente abaixo.

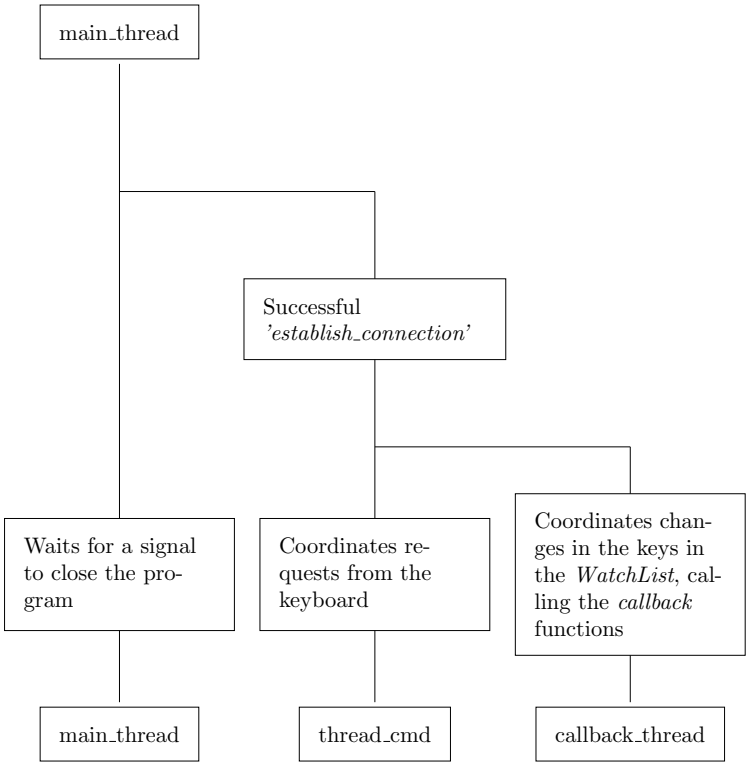

Figura 3: Esquema de criação das *threads* na aplicação.

A *thread callback_thread* tem uma funcionalidade extra que não está presente no esquema. O servidor local pode querer forçar a aplicação a fechar a conexão por uma de duas razões: ou o grupo ao qual a aplicação está ligada foi eliminado, ou o servidor local pode querer terminar o programa quando existem conexões ainda em curso. Nestes dois casos é necessário enviar uma *flag* para a aplicação para a forçar a desligar-se. Visto que não queremos interferir com outras operações que possam estar a ocorrer em simultâneo e visto que não é possível a aplicação saber quando é que tem de receber esta *flag*, aproveitámos o facto de existir uma outra *thread* que está constantemente a receber mensagens através de uma conexão diferente daquela que é utilizada para todas as outras funções. Assim, se nesta *thread* for recebida a *flag* a 1, a aplicação sabe que houve uma chave que foi alterada que pertence à *WatchList*; se for recebida a -1, a *thread* chama a função que fecha a conexão com o servidor local.

As *threads* podem ser destruídas de duas formas, utilizando uma *flag* (inteiro) *exit_flag* para as distinguir:

- A conexão é fechada pelo utilizador, introduzindo o pedido *'close_connection'* no teclado. Neste caso é utilizado o comando *pthread_cancel* para cancelar a *callback_thread* e a *exit_flag* é colocada a 1, indicando à *thread main_thread* que o programa pode finalizar.
- A conexão é fechada pelo servidor, enviando uma *flag* para a *thread callback_thread*. Neste caso é utilizado o comando *pthread_cancel* para cancelar a *thread_cmd* e a *exit_flag* é novamente colocada a 1.

Colocando a *exit_flag* a 1, a *thread main_thread* percebe que tem de acabar o problema.

KVS-LocalServer.c

O servidor local tem uma *thread* principal (*main_thread*) responsável pela conexão ao servidor autenticador para que se possam criar / destruir os grupos, após a qual cria uma *thread handle_apps*, responsável por aceitar as conexões das aplicações. Por cada aplicação que se liga é criada uma nova *thread* (*app_thread*). De seguida, a *thread* principal cria uma *thread thread_cmd* responsável por receber os pedidos através do teclado. Esta cria outra *thread*, *thread_close*, que cria uma nova ligação com o servidor autenticador, independente daquela já criada. Esta *thread* foi criada para, no caso do servidor autenticador se desconectar com o servidor local ainda conectado, este último receber uma mensagem nesta *thread* a forçar o fecho da conexão com o servidor autenticador. Para não haver interferência com as operações de criação e destruição de grupos foi mesmo necessário a criação de um novo *file descriptor* e de um novo endereço com um porto diferente. A outra forma de fechar o servidor local é inserindo o comando "Quit"no teclado. Estas duas formas mudam a variável global *exit_flag*, que indica à *main_thread* que deve acabar o programa.

Note-se também que se quem fechar a conexão for a *thread_close* após receber essa indicação do servidor autenticador, esta utiliza o comando *pthread_cancel* para terminar a *thread_cmd*, ou vice-versa. As *app_thread* são fechadas ou porque as aplicações já se desconectaram ou, como já foi explicado anteriormente, o servidor local força as mesmas a desconectarem-se. Por fim, a *thread handle_apps* é fechada pela *main_thread*. Assim, todas as *threads* são fechadas corretamente e a memória é libertada.

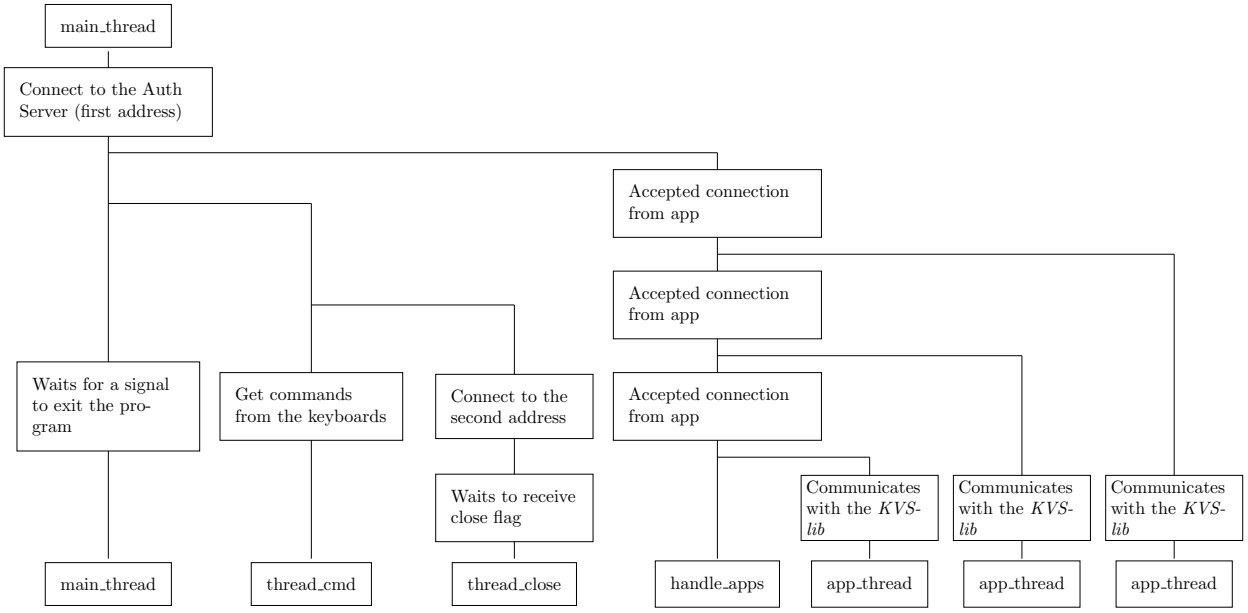

Figura 4: Esquema de criação das *threads* no servidor local.

KVS-AuthServer.c

O mecanismo desde servidor é semelhante ao do servidor local. A única diferença reside na forma como as ligações são feitas. Tendo em conta que o tipo de *socket* é *UDP*, não é possível os servidores locais ligarem-se usando o *connect* e o servidor autenticador aceitar as ligações usando o comando *accept*. Para conseguir lidar com várias ligações em simultâneo, e como cada pedido de um servidor local (para criar um grupo por exemplo) contém vários passos, foi necessário criar um *connect* artificial. Para isto temos uma *thread handle_local_servers* que, ao receber uma mensagem com a *string* "Connect", envia para o servidor local de onde recebeu a mensagem o número do porto que vai ser específico para este servidor. Os dois servidores criam novas *sockets* e assim conseguem comunicar entre si sem interferir nas conexões dos outros servidores locais que possam estar conectados. Devido à possibilidade do servidor autenticador se desconectar antes dos servidores locais, como foi explicado anteriormente, é enviada uma mensagem para cada um dos servidores locais a forçar o fecho da conexão. Para isto é necessário uma segunda "ligação", específica para cada servidor local. Isto foi feito criando uma segunda *thread* para cada servidor conectado (*lserver_close*).

Apresenta-se em baixo o esquema de criação das *threads*.

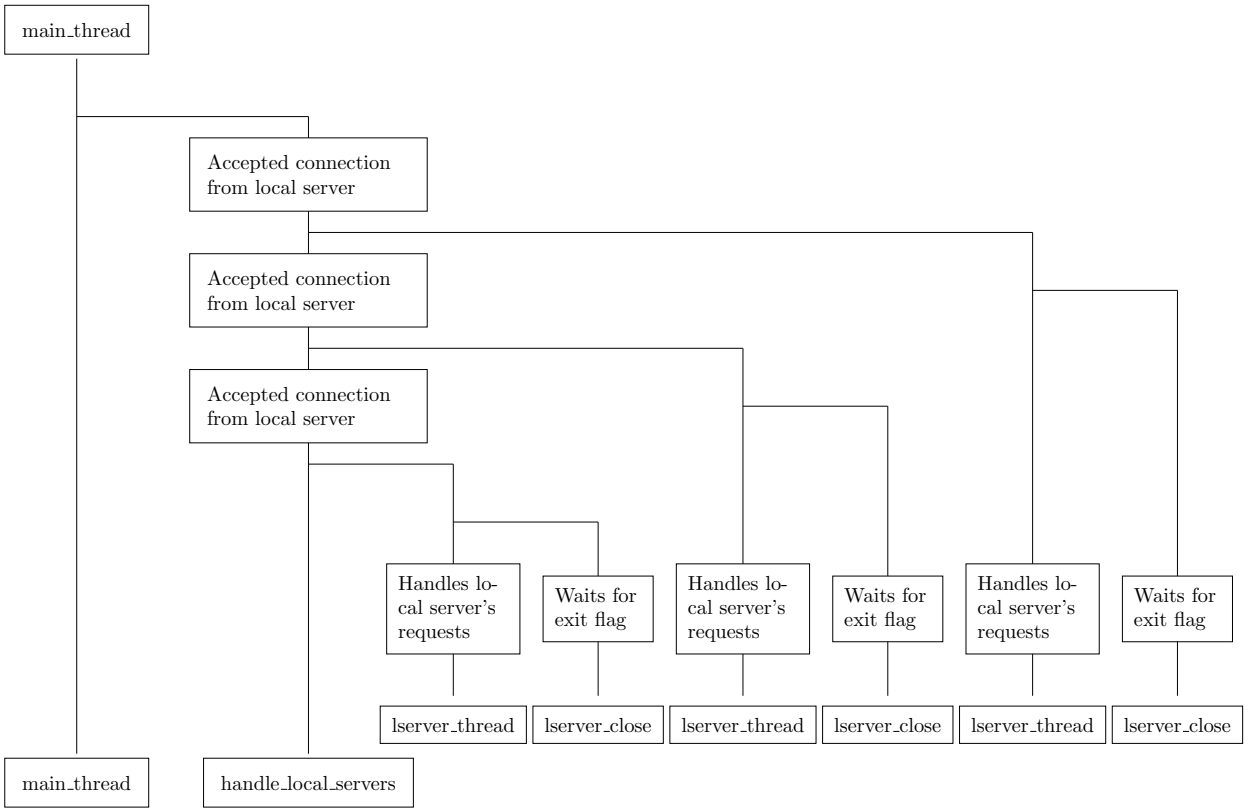

Figura 5: Esquema de criação das *threads* no servidor autenticador.

Comunicação

Diagramas de comunicação entre a aplicação e o servidor local

Começamos por descrever a forma como a *thread thread_cmd*, representada na figura 3, comunica com o terminal / teclado para receber os pedidos do utilizador através do seguinte esquema:

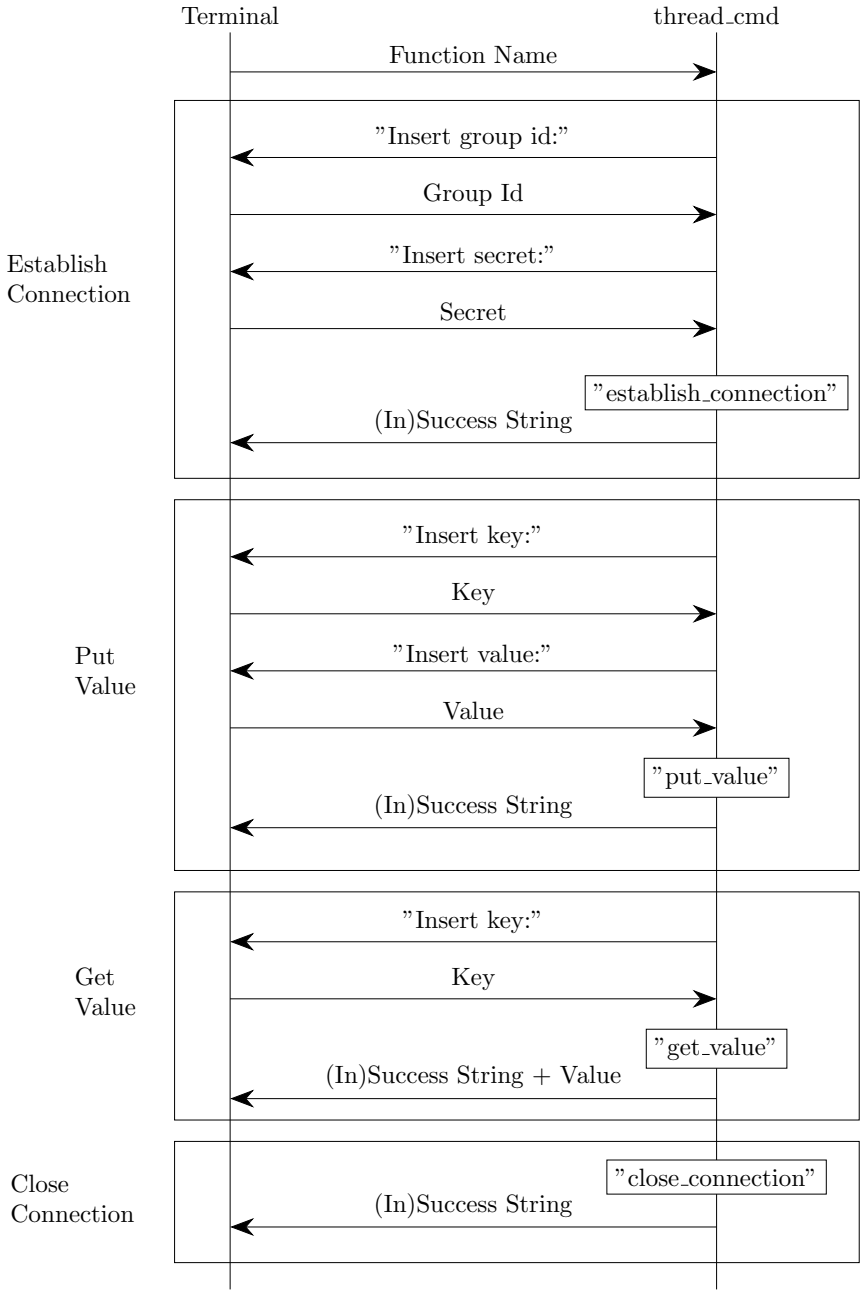

Figura 6: Comunicação entre o terminal / teclado e a aplicação (Parte 1).

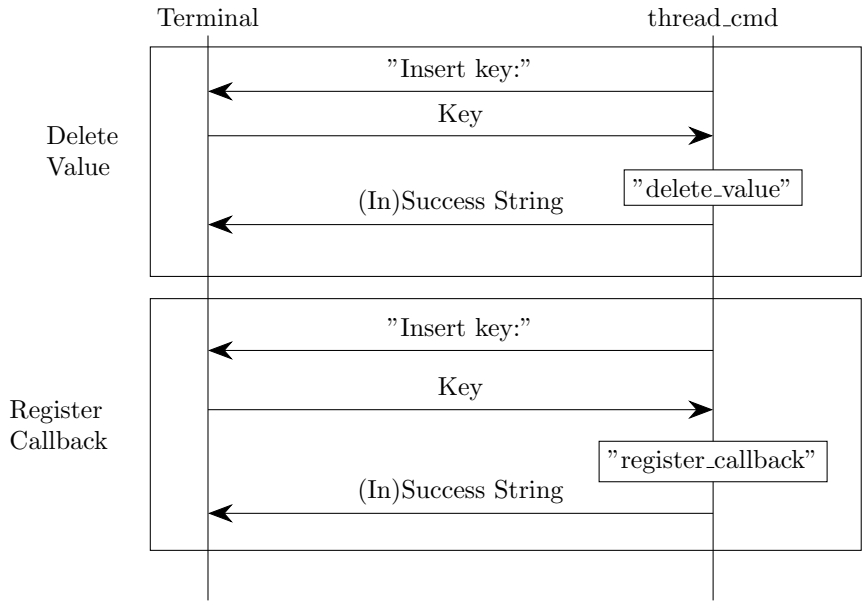

Figura 7: Comunicação entre o terminal / teclado e a aplicação (Parte 2).

Temos então um ciclo onde se recebe o nome da função a executar através do teclado. Dependendo desta, a aplicação pede diferentes argumentos e executa uma função da biblioteca *KVS-lib* diferente.

Iremos agora descrever em detalhe a forma como as funções são implementadas no ficheiro *KVS-li.c*. Note-se que nos esquemas, os nomes a sublinhado são funções a ser executadas pelo servidor local, implementadas no ficheiro *groupList.c*. Todas as operações começam com o envio da aplicação para o servidor local de um inteiro (*Function Code*, que vai de 0 a 4) para que o servidor saiba que conjunto de operações executar.

establish_connection

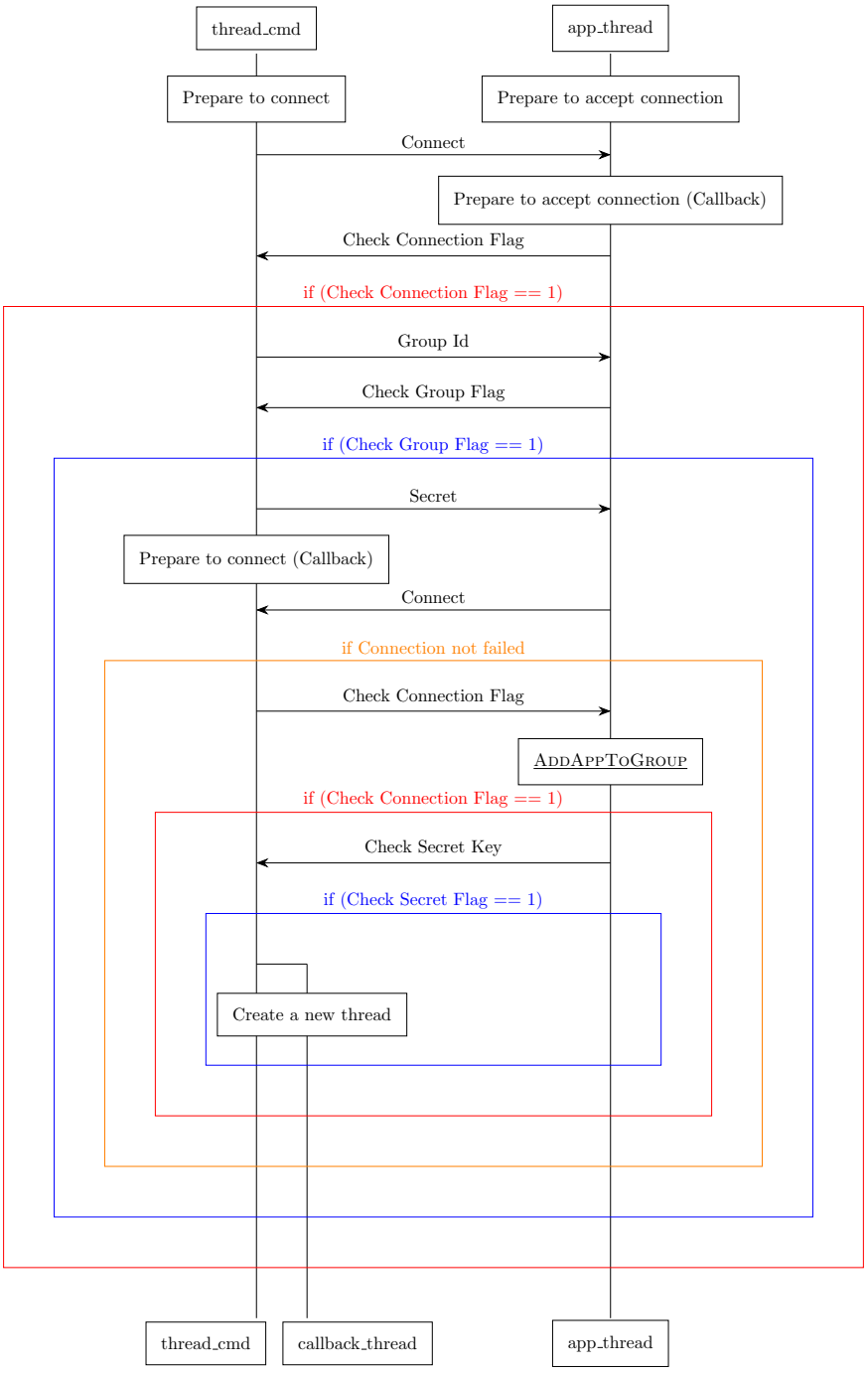

Figura 8: Comunicação entre a biblioteca e o servidor local durante a função *establish_connection*.

Note-se que o *Check Group Flag* diz à aplicação se o grupo ao qual se está a tentar conectar existe ou não. O *Check Key Flag* diz à aplicação se o segredo está ou não correto.

Note-se também que, se algum dos passos falhar, ou seja, se alguma das comunicações resultar em erro ou se as condições *if* descritas não forem satisfeitas, para além de retornar os códigos de erro, o(s) *file descriptor(s)* que foram criados são fechados para se poder tentar realizar uma nova conexão.

Também decidimos que uma aplicação apenas pode estar conectada a um grupo, por isso, se se tentar realizar esta operação pela segunda vez, sendo que a primeira foi bem sucedida, irá resultar em erro.

put_value

Ao colocar um novo valor na tabela, decidimos que, se o utilizador introduzir uma chave que já existe, o valor dessa chave é substituído por aquele que o utilizador der.

Também é importante ter em conta que, se a chave introduzida estiver presente na *WatchList* de qualquer uma das aplicações que esteja ligada ao grupo em questão, é necessário notificar todas elas e não apenas aquela que realizou o pedido para colocar o par chave-valor. Assim, no esquema abaixo, a caixa a azul é realizada para todas as aplicações ainda conectadas com a chave em questão na sua *WatchList*.

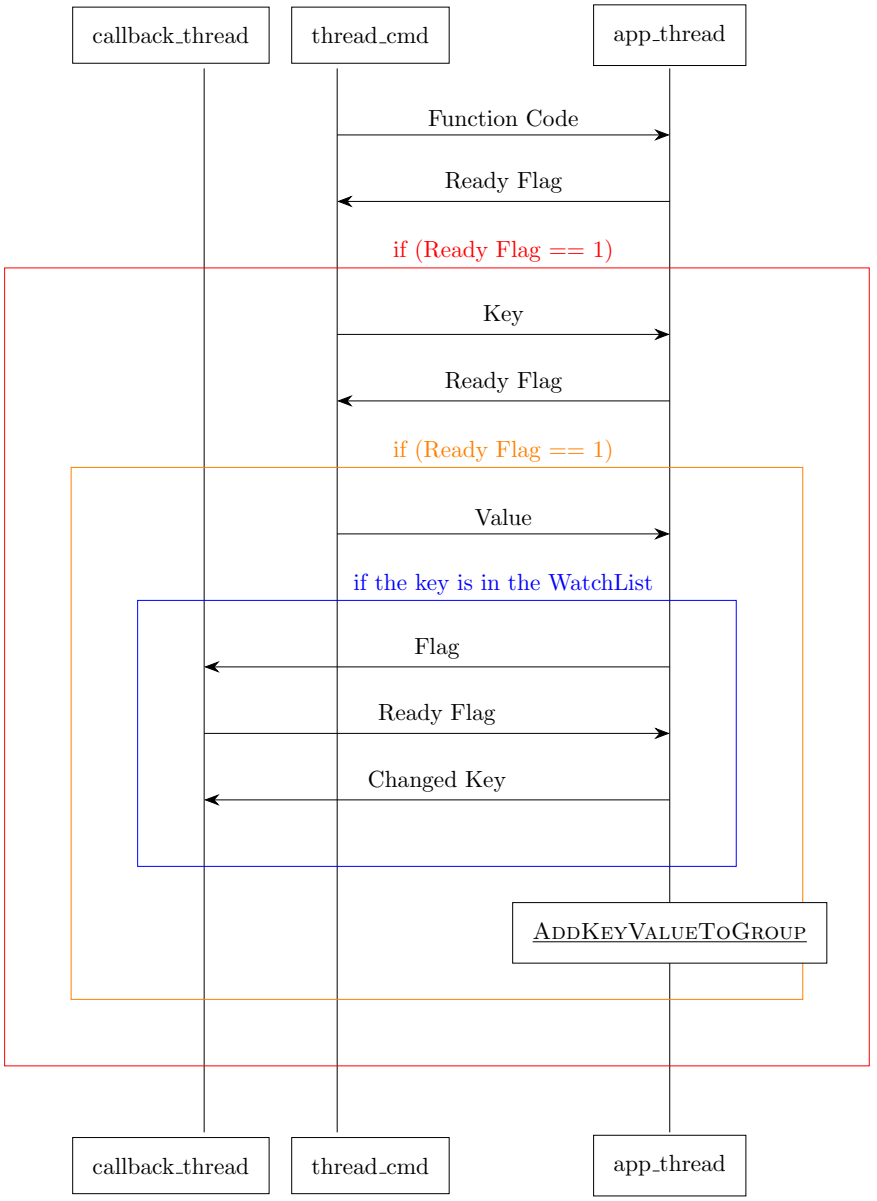

Figura 9: Comunicação entre a biblioteca e o servidor local durante a função *put_value*.

Para verificar se a chave está presente em qualquer uma das *WatchLists* das aplicações que estão conectadas (condição a azul) foram utilizadas funções implementadas no ficheiro *groupList.c*, nomeadamente as funções FindKeyValueLocalServer e IsWatchListOfGroup.

get_value

Para receber o valor para uma dada chave foi necessário enviar o comprimento desse valor antes de enviar o mesmo por duas razões. Primeiro, para a aplicação poder alocar a memória exata para poder receber o valor. Segundo, porque podemos colocar o comprimento com um número negativo para indicar à aplicação que a chave que o utilizador introduziu não existe na tabela do grupo o qual a aplicação pertence.

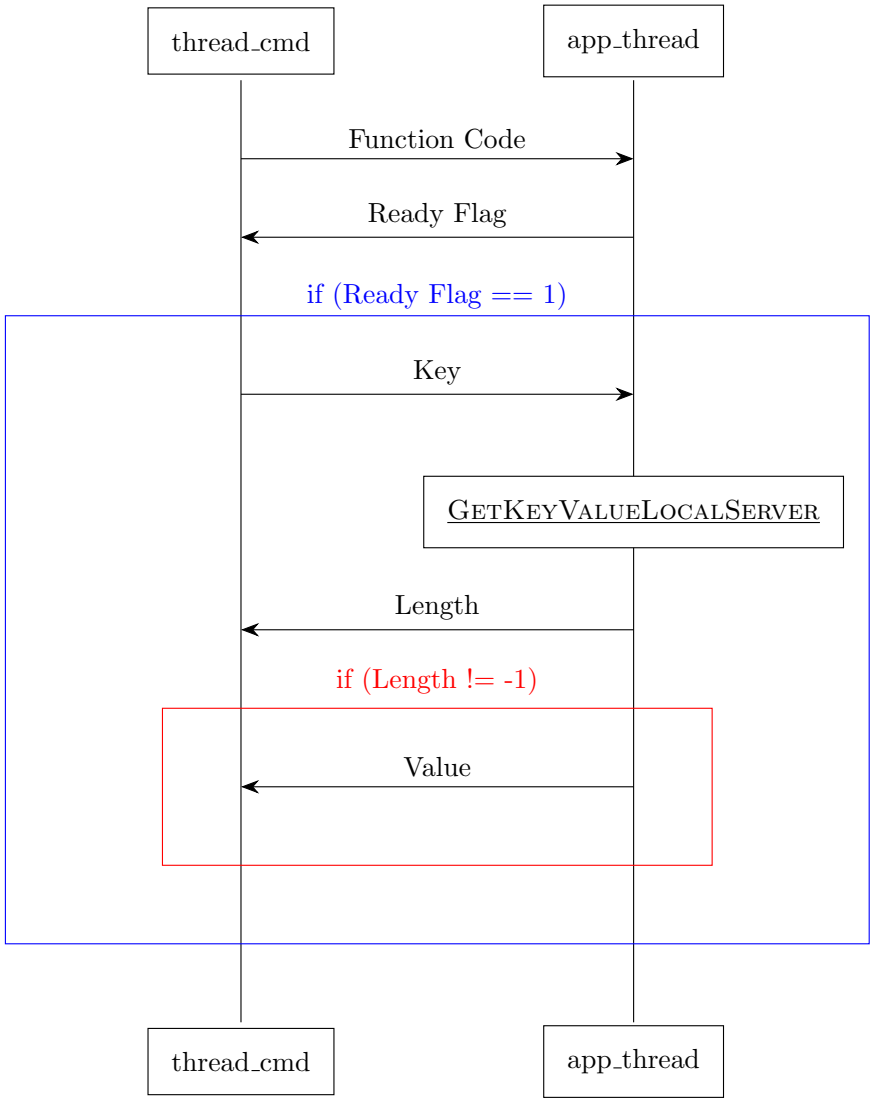

Figura 10: Comunicação entre a biblioteca e o servidor local durante a função *get_value*.

delete_value

Na eliminação de um par chave-valor é preciso também eliminá-lo das *WatchList*, quer da aplicação, quer do servidor local.

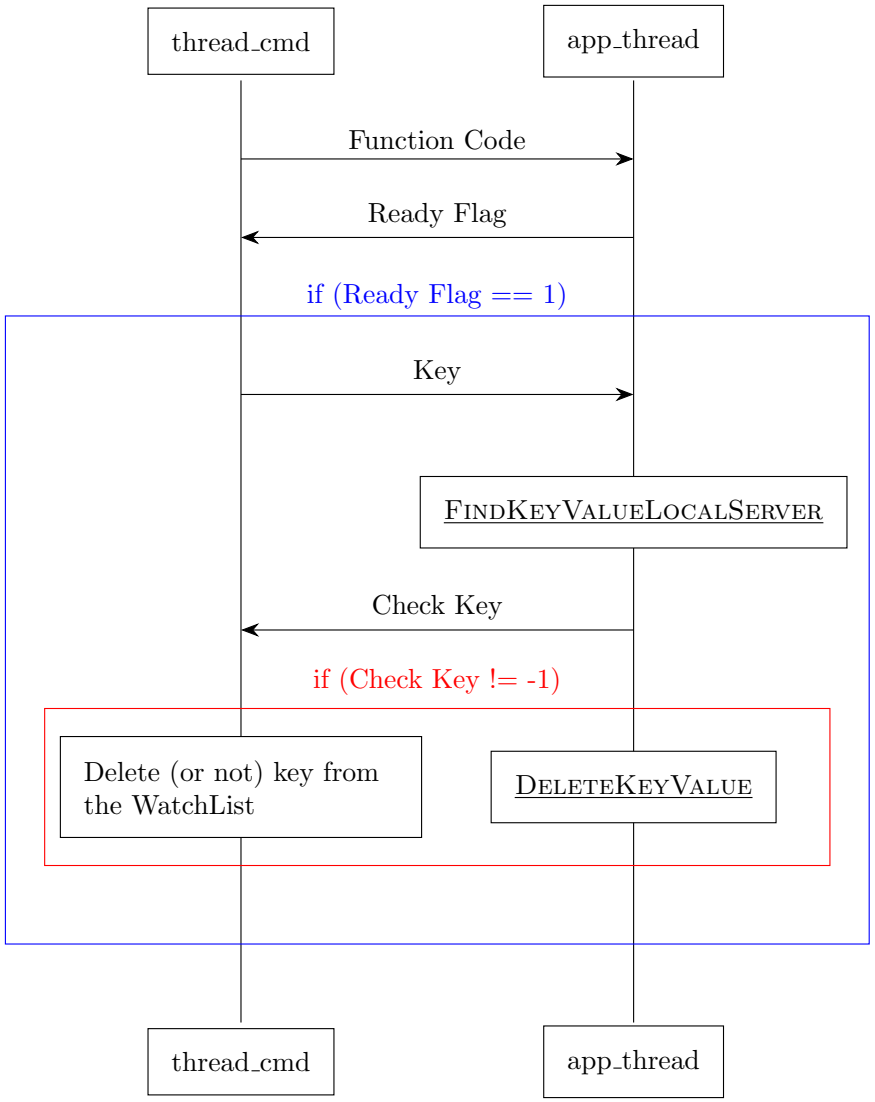

Figura 11: Comunicação entre a biblioteca e o servidor local durante a função *delete_value*.

register_callback

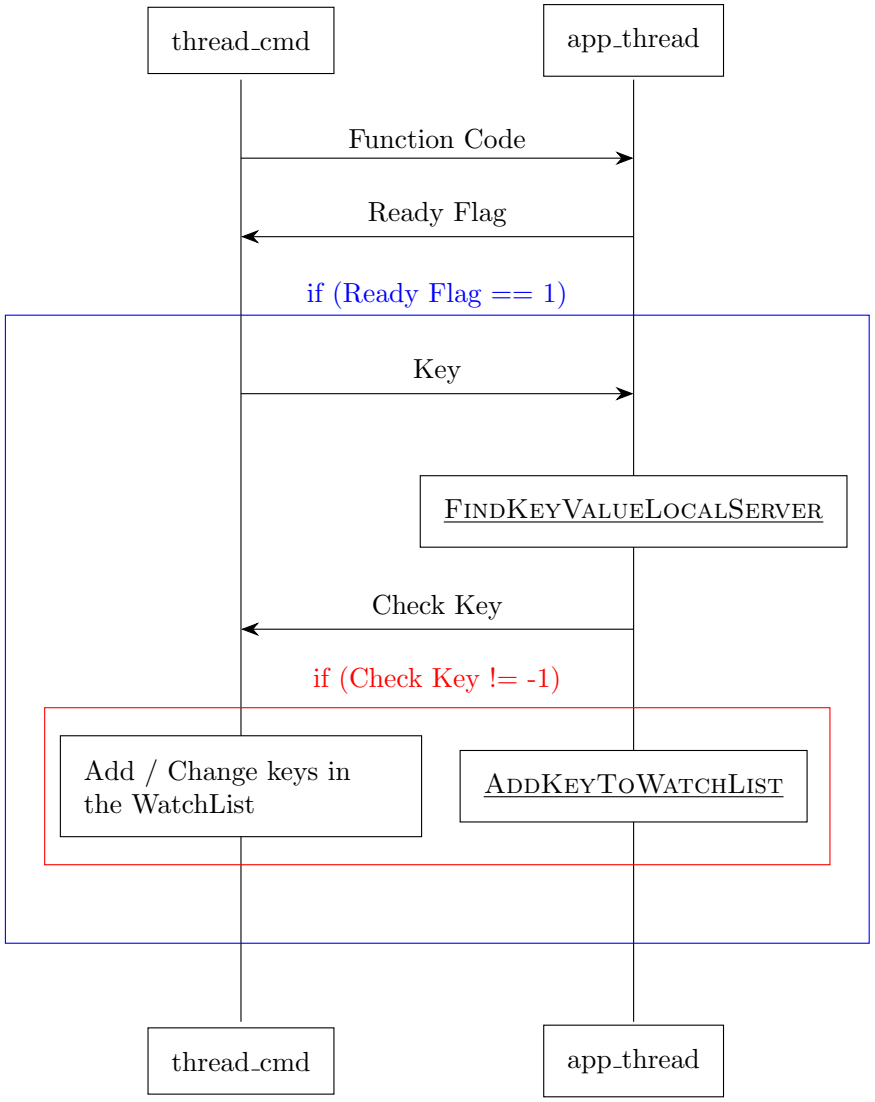

Figura 12: Comunicação entre a biblioteca e o servidor local durante a função *register_callback*.

Sobre o esquema acima, a variável *Check Key* diz à aplicação se a chave que enviou está ou não presente na tabela do grupo ao qual está ligada, visto que só faz sentido registar uma função numa chave que existe.

close_connection

Ao fechar a conexão com o servidor local é importante eliminar a *WatchList* para que a memória alocada seja libertada antes do programa acabar.

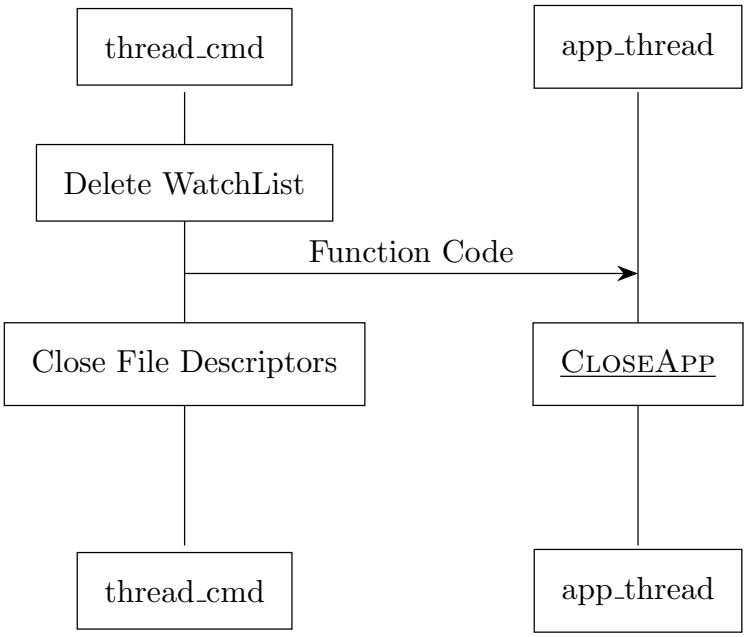

Figura 13: Comunicação entre a biblioteca e o servidor local durante a função *close_connection*.

Códigos de erro

Na biblioteca *KVS-lib* foram implementados vários códigos de erro que são impressos em caso de insucesso de alguma das operações apresentadas anteriormente.

Retorno em caso de erro	Tipo de erro
-1	Erro ao fechar a socket
-2	Erro a criar a socket ou socket não criada
-3	Erro ao enviar o código da função
-4	Erro ao receber a *ready flag*
-5	Servidor não pronto (*ready flag* != 1)
-6	Erro ao enviar chave / nome do grupo / valor / segredo
-7	Chave ou nome do grupo não existente
-8	Erro ao receber resposta para chave / nome do grupo / segredo enviado
-9	Segredo incorreto
-10	Erro ao receber valor
-11	Erro no *binding*
-12	Erro no *connect*
-13	Erro ao receber a resposta para a conexão estabelecida
-14	Erro em estabelecer conexão com o servidor local
-15	Outro erro
-16	Conexão já estabelecida

Tabela 1: Código de erro utilizados na biblioteca *KVS-lib*.

Diagramas de comunicação entre o servidor local e o servidor autenticador

Nesta secção está descrita a comunicação entre o servidor local e o servidor autenticador para as operações que o utilizador pode pedir através do teclado. Para cada operação existem dois esquemas: um primeiro esquema com a interação entre o terminal / teclado e o servidor local (semelhante aos das figuras 6 e 7), e um segundo esquema com as interações entre o servidor local e o servidor autenticador. Note-se que as funções a sublinhado no servidor autenticador correspondem a funções do ficheiro auxiliar *hash.c* e as funções a sublinhado no servidor local correspondem a funções do ficheiro auxiliar *groupList.c*.

Create group

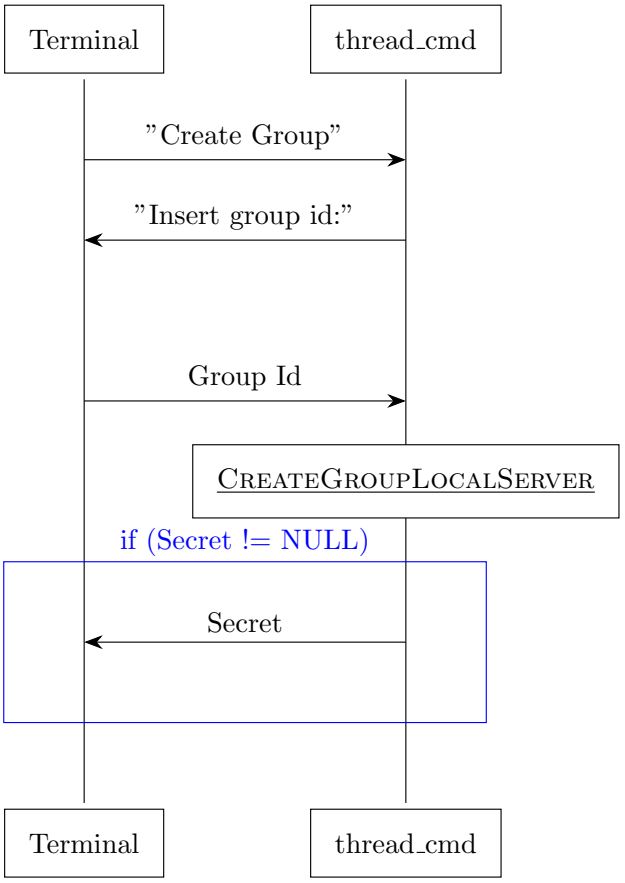

Figura 14: Comunicação entre o terminal e o servidor local aquando da criação de um novo grupo.

Note-se que se o retorno da função *CreateGroupLocalServer* for *NULL* e não existir nenhuma mensagem de erro é porque o grupo que estamos a tentar criar já existe. No próximo esquema apresentamos a arquitectura da função *CreateGroupLocalServer* em detalhe.

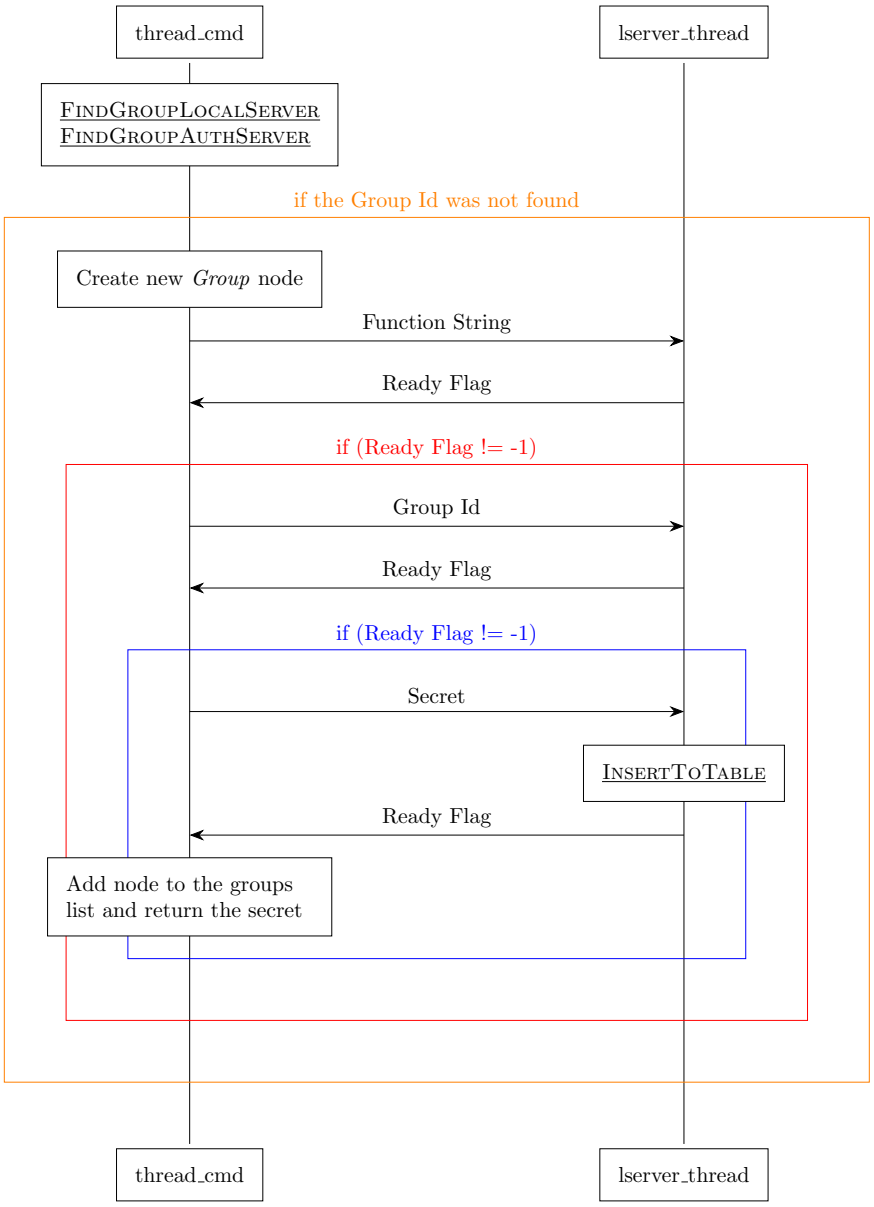

Figura 15: Passos da função CreateGroupLocalServer.

Se as condições *if* não forem satisfeitas a função retorna *NULL*. Decidimos que o segredo gerado é uma string aleatória com 5 caracteres.

Delete group

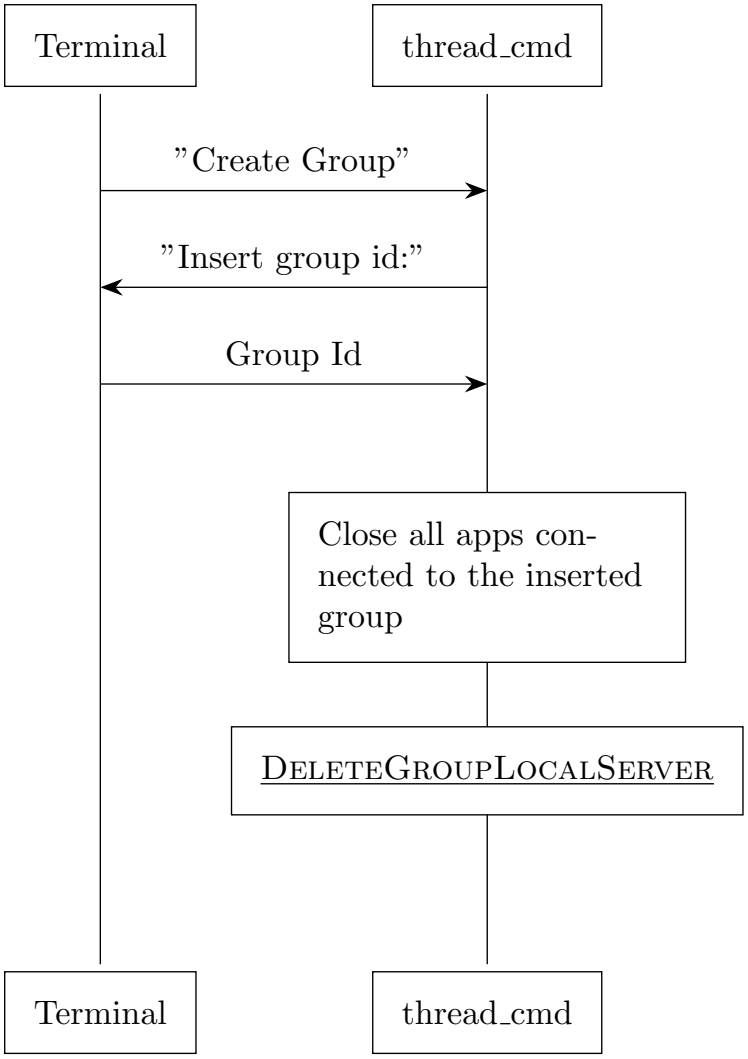

Figura 16: Comunicação entre o terminal e o servidor local aquando da eliminação de um grupo.

Novamente, apresentamos em detalhe a comunicação da função *DeleteGroupLocalServer*.

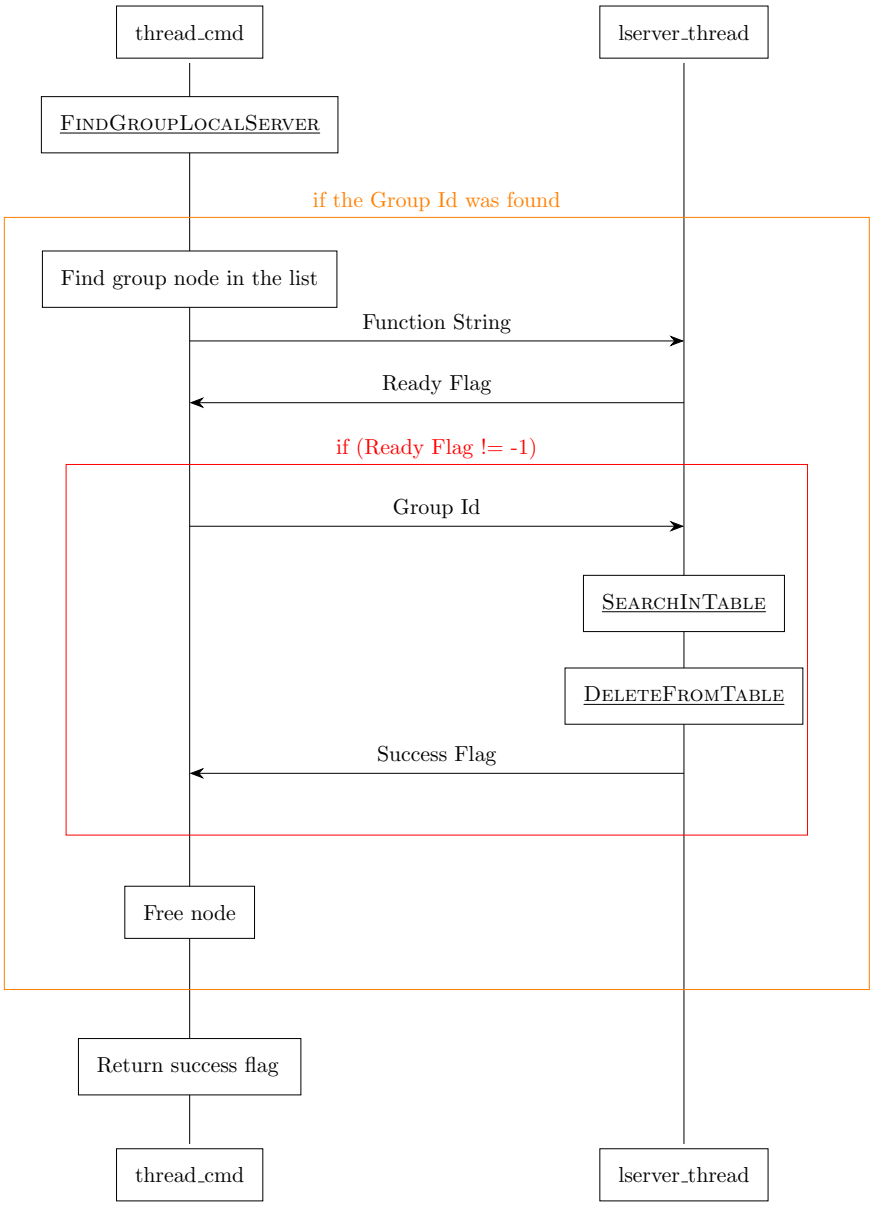

Figura 17: Passos da função DeleteGroupLocalServer.

Show group info

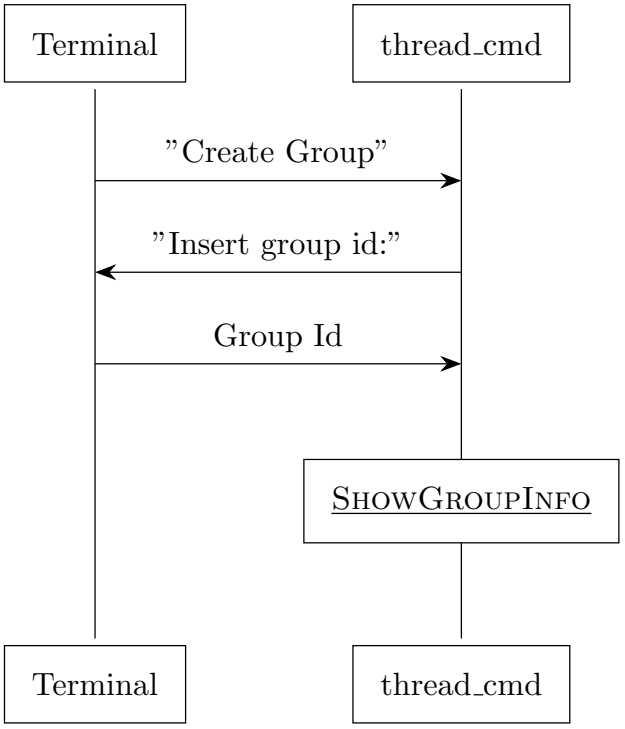

Figura 18: Comunicação entre o terminal e o servidor local aquando da impressão da informação de um grupo.

Esta operação envolve imprimir o segredo de um determinado grupo. É por isso necessário utilizar uma função presente no ficheiro *groupList.c* chamada *GetSecretFromAuthServer* e que envolve comunicar com o servidor autenticador. O esquema da comunicação é apresentado abaixo e note-se que a função *GetSecretFromAuthServer* corresponde às operações dentro do retângulo alaranjado.

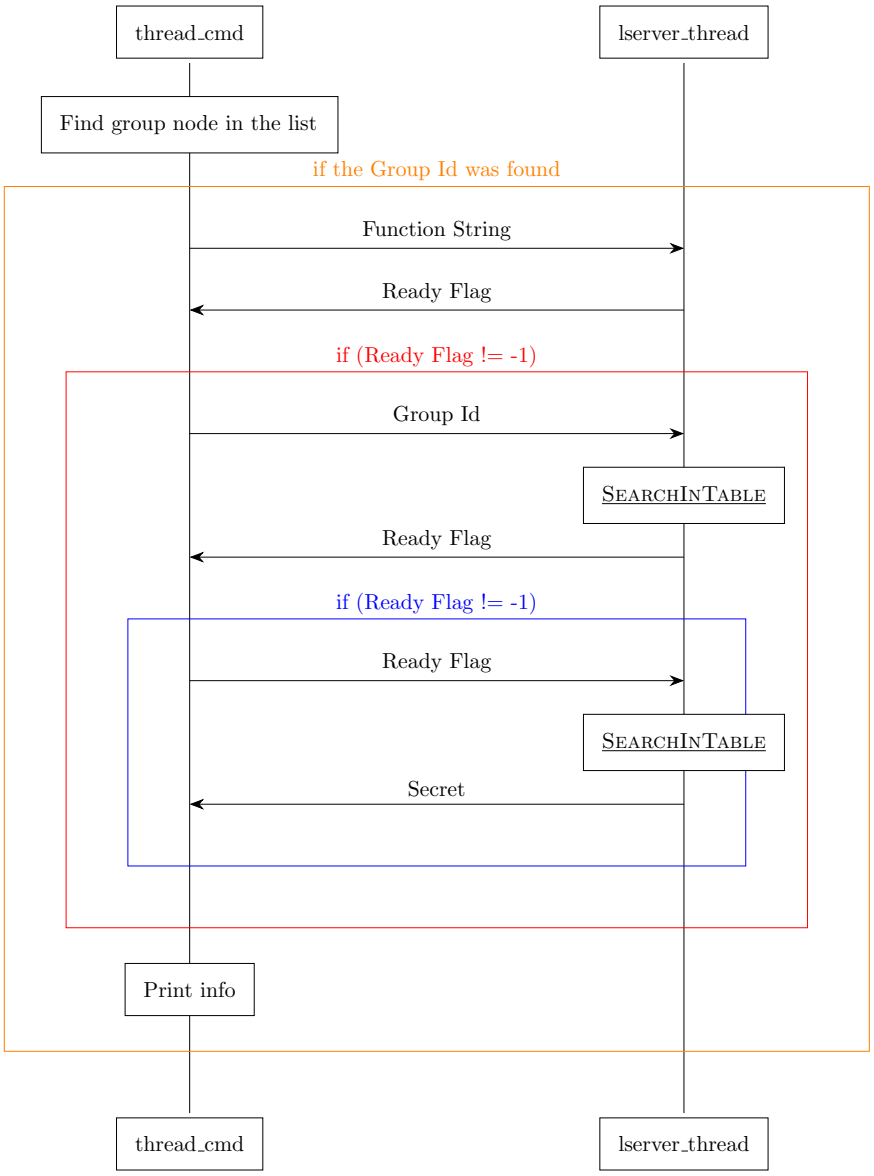

Figura 19: Passos da função ShowGroupInfo.

Show application status

Este comando é extremamente simples visto que não envolve nenhuma comunicação com o servidor autenticador e que basta introduzir no teclado o nome da função para a chamar. Toda a informação necessária está presente na lista de grupos. Para além da informação pedida do enunciado do projeto decidimos imprimir também o grupo a que as aplicações pertencem, o estado da aplicação (ligada ou não ligada), o tempo que a aplicação esteve conectada (ou o tempo desde o início da conexão se a aplicação ainda estiver ligada), e as chaves que têm funções de *callback* associadas em cada aplicação.

Função auxiliar - *FindGroupAuthServer*

Foi criada também uma função auxiliar (utilizada aquando da criação e eliminação de grupos). O seu esquema de comunicação é apresentado abaixo.

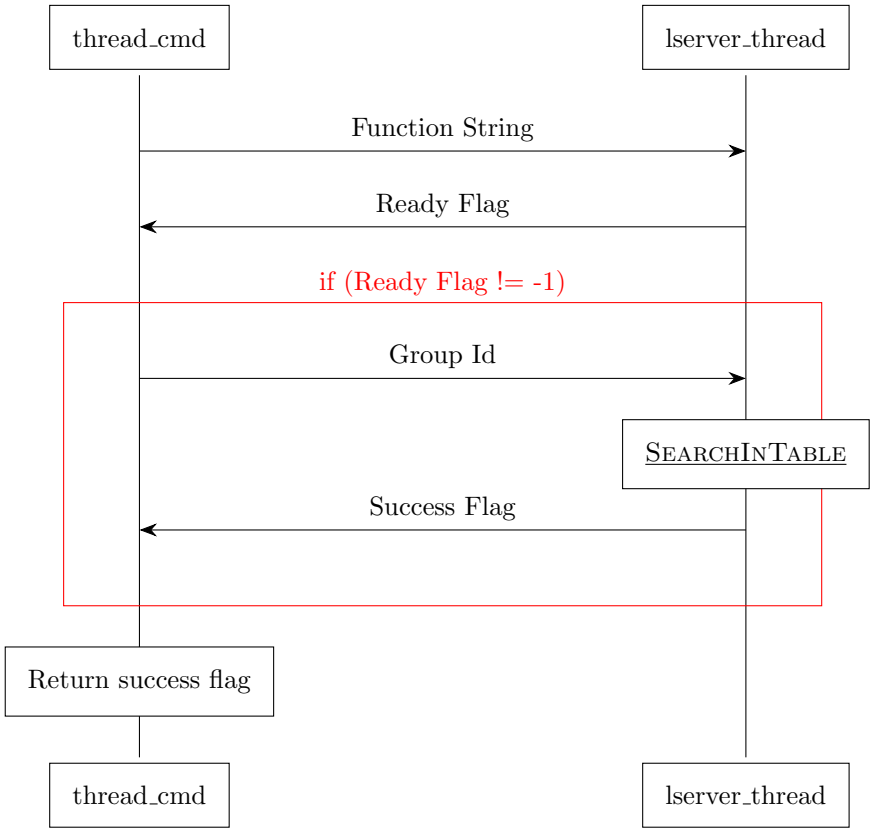

Figura 20: Passos da função FindGroupAuthServer.

Terminar os servidores

Tanto o servidor local como o servidor autenticador podem ser terminados através da inserção do comando "Quit"no teclado.

No caso do servidor local, quando é pedido para este terminar, todas as aplicações ligadas são terminadas, os grupos são eliminados e é enviada uma mensagem para o servidor autenticador a avisar que se vai desligar.

Se for pedido ao servidor autenticador para este se desligar, são enviadas mensagens a todos os servidores locais ligados para que estes possam fechar todas as suas conexões e para todas as estruturas de dados serem corretamente eliminadas e a sua memória libertada.

Sincronização

Em termos de sincronização temos de ter cuidado com várias situações mas chamamos especial atenção às situações descritas de seguida:

- Várias aplicações ligadas ao mesmo grupo de um determinado servidor podem fazer um pedido simultaneamente. Existem casos em que isto não tem problema: por exemplo, se uma aplicação pedir para colocar uma chave e outra pedir para eliminar uma chave e estas tenham índices da tabela diferente, não existe nenhum *race condition*. Mas, se quiserem as duas adicionar valores com a mesma chave, ou se uma quiser adicionar e outra eliminar a mesma chave, é possível que as operações não corram da forma desejada. Decidimos então escolher uma solução simplista. Colocamos um *mutex_lock* no início de cada operação que pode ser requisitada (*put_value*, *delete_value*, etc) e um *mutex_unlock* no

fim. Esta solução torna o sistema mais lento visto que as aplicações têm de esperar umas pelas outras para realizar as operações, mas garantimos que não existem aplicações a alterar o mesmo espaço da memória simultâneamente.

- Uma aplicação pode requisitar um valor de um grupo que pode estar a ser eliminado. Assim, repetimos o procedimento anterior de trancar também as operações que podem ser requisitadas ao servidor local pelo utilizador como a criação e eliminação de grupos.
- À semelhança das aplicações quererem realizar operações em simultâneo no servidor local, também vários servidores locais podem requisitar operações simultâneas ao servidor autenticador. Por exemplo, um grupo pode querer ser simultaneamente criado e eliminado. Assim, à semelhança do que foi feito anteriormente, colocamos um *mutex_lock* e um *mutex_unlock* entre cada uma das operações que o servidor autenticador pode realizar.

Note-se que estamos a "trancar"as operações em si que consistem em vários passos e não apenas os passos onde fazemos acesso a memória.

Outros possíveis erros

Para além das várias situações descritas, como por exemplo um servidor local eliminar um grupo ao qual uma aplicação está ligada, existem outras situações que podem induzir em erros que comprometam todo o sistema. Uma delas é se por acaso o servidor autenticador terminar abruptamente (p.e. com CTRL-C). Se isto acontecer e o servidor local tentar realizar uma operação como por exemplo criar um grupo, ficaria preso infinitamente num *sendto* a tentar comunicador com o servidor autenticador. A solução que encontrámos para este caso foi utilizar a função *setsockopt* para, caso a *socket* que comunica com o servidor autenticador não conseguir realizar o *sendto* em menos de um segundo, o servidor local é terminado e é impresso o erro que resultou da tentativa de comunicação.

Um comportamento semelhante acontece com o servidor local e a aplicação - se o servidor local terminar abruptamente, quando na aplicação se tenta realizar uma operação e esta tem um erro na comunicação com o servidor, a aplicação termina e é impresso o erro retornado pelo *send / recv*.